Gobs of data

Rob Pike
24 March 2011

Introduction

To transmit a data structure across a network or to store it in a file, it must be encoded and then decoded again. There are many encodings available, of course: JSON, XML, Google's protocol buffers, and more. And now there's another, provided by Go's gob package.

Why define a new encoding? It's a lot of work and redundant at that. Why not just use one of the existing formats? Well, for one thing, we do! Go has packages supporting all the encodings just mentioned (the protocol buffer package is in a separate repository but it's one of the most frequently downloaded). And for many purposes, including communicating with tools and systems written in other languages, they're the right choice.

But for a Go-specific environment, such as communicating between two servers written in Go, there's an opportunity to build something much easier to use and possibly more efficient.

Gobs work with the language in a way that an externally-defined, language-independent encoding cannot. At the same time, there are lessons to be learned from the existing systems.

Goals

The gob package was designed with a number of goals in mind.

First, and most obvious, it had to be very easy to use. First, because Go has reflection, there is no need for a separate interface definition language or “protocol compiler”. The data structure itself is all the package should need to figure out how to encode and decode it. On the other hand, this approach means that gobs will never work as well with other languages, but that's OK: gobs are unashamedly Go-centric.

Efficiency is also important. Textual representations, exemplified by XML and JSON, are too slow to put at the center of an efficient communications network. A binary encoding is necessary.

Gob streams must be self-describing. Each gob stream, read from the beginning, contains sufficient information that the entire stream can be parsed by an agent that knows nothing a priori about its contents. This property means that you will always be able to decode a gob stream stored in a file, even long after you've forgotten what data it represents.

There were also some things to learn from our experiences with Google protocol buffers.

Protocol buffer misfeatures

Protocol buffers had a major effect on the design of gobs, but have three features that were deliberately avoided. (Leaving aside the property that protocol buffers aren't self-describing: if you don't know the data definition used to encode a protocol buffer, you might not be able to parse it.)

First, protocol buffers only work on the data type we call a struct in Go. You can't encode an integer or array at the top level, only a struct with fields inside it. That seems a pointless restriction, at least in Go. If all you want to send is an array of integers, why should you have to put it into a struct first?

Next, a protocol buffer definition may specify that fields `T.x` and `T.y` are required to be present whenever a value of type `T` is encoded or decoded. Although such required fields may seem like a good idea, they are costly to implement because the codec must maintain a separate data structure while encoding and decoding, to be able to report when required fields are missing. They're also a maintenance problem. Over time, one may want to modify the data definition to remove a required field, but that may cause existing clients of the data to crash. It's better not to have them in the encoding at all. (Protocol buffers also have optional fields. But if we don't have required fields, all fields are optional and that's that. There will be more to say about optional fields a little later.)

The third protocol buffer misfeature is default values. If a protocol buffer omits the value for a "defaulted" field, then the decoded structure behaves as if the field were set to that value. This idea works nicely when you have getter and setter methods to control access to the field, but is harder to handle cleanly when the container is just a plain idiomatic struct. Required fields are also tricky to implement: where does one define the default values, what types do they have (is text UTF-8? uninterpreted bytes? how many bits in a float?) and despite the apparent simplicity, there were a number of complications in their design and implementation for protocol buffers. We decided to leave them out of gobs and fall back to Go’s trivial but effective defaulting rule: unless you set something otherwise, it has the “zero value” for that type - and it doesn’t need to be transmitted.

So gobs end up looking like a sort of generalized, simplified protocol buffer. How do they work?

Values

The encoded gob data isn't about types like `int8` and `uint16`. Instead, somewhat analogous to constants in Go, its integer values are abstract, sizeless numbers, either signed or unsigned. When you encode an `int8`, its value is transmitted as an unsized, variable-length integer. When you encode an `int64`, its value is also transmitted as an unsized, variable-length integer. (Signed and unsigned are treated distinctly, but the same unsized-ness applies to unsigned values too.) If both have the value 7, the bits sent on the wire will be identical. When the receiver decodes that value, it puts it into the receiver's variable, which may be of arbitrary integer type. Thus an encoder may send a 7 that came from an `int8`, but the receiver may store it in an `int64`. This is fine: the value is an integer and as a long as it fits, everything works. (If it doesn't fit, an error results.) This decoupling from the size of the variable gives some flexibility to the encoding: we can expand the type of the integer variable as the software evolves, but still be able to decode old data.

This flexibility also applies to pointers. Before transmission, all pointers are flattened. Values of type `int8`, `*int8`, `**int8`, `****int8`, etc. are all transmitted as an integer value, which may then be stored in `int` of any size, or `*int`, or `******int`, etc. Again, this allows for flexibility.

Flexibility also happens because, when decoding a struct, only those fields that are sent by the encoder are stored in the destination. Given the value

```
type T struct{ X, Y, Z int } // Only exported fields are encoded
and decoded.
```

```
and decoded:
var t = T{X: 7, Y: 0, Z: 8}
```

the encoding of `t` sends only the 7 and 8. Because it's zero, the value of `Y` isn't even sent; there's no need to send a zero value.

The receiver could instead decode the value into this structure:

```
type U struct{ X, Y *int8 } // Note: pointers to int8s
var u U
```

and acquire a value of `u` with only `X` set (to the address of an `int8` variable set to 7); the `Z` field is ignored - where would you put it? When decoding structs, fields are matched by name and compatible type, and only fields that exist in both are affected. This simple approach finesses the “optional field” problem: as the type `T` evolves by adding fields, out of date receivers will still function with the part of the type they recognize. Thus gobs provide the important result of optional fields - extensibility - without any additional mechanism or notation.

From integers we can build all the other types: bytes, strings, arrays, slices, maps, even floats. Floating-point values are represented by their IEEE 754 floating-point bit pattern, stored as an integer, which works fine as long as you know their type, which we always do. By the way, that integer is sent in byte-reversed order because common values of floating-point numbers, such as small integers, have a lot of zeros at the low end that we can avoid transmitting.

One nice feature of gobs that Go makes possible is that they allow you to define your own encoding by having your type satisfy the GobEncoder and GobDecoder interfaces, in a manner analogous to the JSON package's Marshaler and Unmarshaler and also to the Stringer interface from package fmt. This facility makes it possible to represent special features, enforce constraints, or hide secrets when you transmit data. See the documentation for details.

Types on the wire

The first time you send a given type, the gob package includes in the data stream a description of that type. In fact, what happens is that the encoder is used to encode, in the standard gob encoding format, an internal struct that describes the type and gives it a unique number. (Basic types, plus the layout of the type description structure, are predefined by the software for bootstrapping.) After the type is described, it can be referenced by its type number.

Thus when we send our first type `T`, the gob encoder sends a description of `T` and tags it with a type number, say 127. All values, including the first, are then prefixed by that number, so a stream of `T` values looks like:

```
("define type id" 127, definition of type T)(127, T value)(127, T value), ...
```

These type numbers make it possible to describe recursive types and send values of those types. Thus gobs can encode types such as trees:

```
type Node struct {
    Value       int
```

```
    Value       int
    Left, Right *Node
}
```

(It’s an exercise for the reader to discover how the zero-defaulting rule makes this work, even though gobs don’t represent pointers.)

With the type information, a gob stream is fully self-describing except for the set of bootstrap types, which is a well-defined starting point.

Compiling a machine

The first time you encode a value of a given type, the gob package builds a little interpreted machine specific to that data type. It uses reflection on the type to construct that machine, but once the machine is built it does not depend on reflection. The machine uses package unsafe and some trickery to convert the data into the encoded bytes at high speed. It could use reflection and avoid unsafe, but would be significantly slower. (A similar high-speed approach is taken by the protocol buffer support for Go, whose design was influenced by the implementation of gobs.) Subsequent values of the same type use the already-compiled machine, so they can be encoded right away.

[Update: As of Go 1.4, package unsafe is no longer use by the gob package, with a modest performance drop.]

Decoding is similar but harder. When you decode a value, the gob package holds a byte slice representing a value of a given encoder-defined type to decode, plus a Go value into which to decode it. The gob package builds a machine for that pair: the gob type sent on the wire crossed with the Go type provided for decoding. Once that decoding machine is built, though, it's again a reflectionless engine that uses unsafe methods to get maximum speed.

Use

There's a lot going on under the hood, but the result is an efficient, easy-to-use encoding system for transmitting data. Here's a complete example showing differing encoded and decoded types. Note how easy it is to send and receive values; all you need to do is present values and variables to the gob package and it does all the work.

```
package main
```

```
import (
    "bytes"
    "encoding/gob"
    "fmt"
    "log"
)

type P struct {
    X, Y, Z int
    Name    string
}

type Q struct {
    X, Y *int32
    Name string
}

func main() {
    // Initialize the encoder and decoder.  Normally enc and dec
would be
```

```
    // bound to network connections and the encoder and decoder w
ould
    // run in different processes.
    var network bytes.Buffer        // Stand-in for a network con
nection
    enc := gob.NewEncoder(&network) // Will write to network.
    dec := gob.NewDecoder(&network) // Will read from network.
    // Encode (send) the value.
    err := enc.Encode(P{3, 4, 5, "Pythagoras"})
    if err != nil {
        log.Fatal("encode error:", err)
    }
    // Decode (receive) the value.
    var q Q
    err = dec.Decode(&q)
    if err != nil {
        log.Fatal("decode error:", err)
    }
    fmt.Printf("%q: {%d,%d}\n", q.Name, *q.X, *q.Y)
}
```

You can compile and run this example code in the Go Playground.

The rpc package builds on gobs to turn this encode/decode automation into transport for method calls across the network. That's a subject for another article.

Details

The gob package documentation, especially the file doc.go, expands on many of the details described here and includes a full worked example showing how the encoding represents data. If you are interested in the innards of the gob implementation, that's a good place to start.

Next article: Godoc: documenting Go code
Previous article: C? Go? Cgo!
Blog Index